

```
/* modiSBML2.c

Copyright (c) 2007-2010. Free Software Foundation, Inc.

This file is part of GNU MCSim.

GNU MCSim is free software; you can redistribute it and/or
modify it under the terms of the GNU General Public License
as published by the Free Software Foundation; either version 3
of the License, or (at your option) any later version.

GNU MCSim is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
GNU General Public License for more details.

You should have received a copy of the GNU General Public License
along with GNU MCSim; if not, see <http://www.gnu.org/licenses/>

-- Revisions -----
Logfile: %F%
Revision: %I%
Date: %G%
Modtime: %U%
Author: @a
-- SCCS -----

Handles input parsing of the SBML Model Definition Files and of the
template model (if used).
Requires libSBML to be installed.
*/
#include "config.h"

/* if config.h defines HAVE_LIBSBML this file is compiled */

#ifndef HAVE_LIBSBML

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <stdarg.h>
#include <assert.h>

#include "lexerr.h"
#include "mod.h"
#include "modi.h"
#include "modiSBML.h"
#include "modd.h"
#include "modo.h"
#include "strutil.h"

#include "sbml/SBMLReader.h"
#include "sbml/SBMLTypes.h"
```

```

/* -----
-----
    Private Enums */

typedef enum {
    product,
    reactant,
    modifier,
    parameter
} VARTYPES;

typedef enum {
    plus,
    minus
} OPSIGNS;

typedef enum {
    ID,
    name,
    metaID
} ID_OPTIONS;

/* -----
-----
    Private Typedefs */

/* pack what is needed for SetVar */
typedef struct tagFORSV {
    PINPUTBUF pibIn;
    PSTR szName;
    PSTR szVal;
    PVMMAPSTRCT pTarget;
} FORSV, *PFORSV;

/* -----
-----
    Private Globals */

/* those two will be initialized in ReadOptions: */

ID_OPTIONS iIDtype; /* flag to indicate whether names or IDs should be
used */
PSTRLEX     szDefault_Cpt; /* name of default (external) compartment */

/* -----
-----
    AugmentEquation

Augment the equation field of PVMMAPSTRCT with the equation szEqn by

```



```

PSTRLEX szSName;
PSTRLEX szStoichio;
PVMMAPSTRCT pvm;
PINPUTINFO pinfo = (PINPUTINFO) pibIn->pInfo;

/* get species name */
sprintf (szSName, "%s", SpeciesReference_getSpecies (species));

sprintf (szStoichio, "%g", SpeciesReference_getStoichiometry
(species));

if (!MyStrcmp(szStoichio, "0")) {
    sprintf (szStoichio, "1");
    printf (" %s stoichio was 0, now set to %s\n", szSName, szStoichio);
}
else
    printf (" '%s' (stoichio: %s) ", szSName, szStoichio);

/* reactions are supposed to happen in a defined compartment:
   pad species name with that compartment name */
if (!GetVarPTR (pinfo->pvmGloVars, szSName))
    sprintf (szSName, "%s_%s", szSName, pinfo->pvmLocalCpts->szName);

printf ("processed as '%s'\n", szSName);

/* padded species should have been declared State variables, or
   parameters if they were set to boundary conditions */
hType = GetVarType (pinfo->pvmGloVars, szSName);
if (hType == ID_STATE) { /* deal with differential */
    /* construct equation */
    pvm = GetVarPTR (pinfo->pvmDynEqns, szSName);
    if (!pvm)
        /* no dynamic equation yet defined for szSName, create one */
        DefineVariable (pibIn, szSName, "", KM_DXDT);
    /* pvmDynEqns has been initialized, refresh */
    pvm = GetVarPTR (pinfo->pvmDynEqns, szSName);
}
if (eType != modifier)
    AugmentEquation (pvm, szRName, szStoichio,
                      (eType == reactant ? minus : plus));
}
else {
    if (hType != ID_PARM)
        ReportError (NULL, RE_BADSTATE | RE_FATAL, szSName, NULL);
}

} /* ConstructEqn */

/*
-----
SetVar

Declare a global variable and link its value to it.

```

```

/*
void SetVar (PINPUTBUF pibIn, PSTR szName, PSTR szVal, HANDLE hType)
{
    PVMMAPSTRCT pvm;
    PINPUTINFO pinfo = (PINPUTINFO) pibIn->pInfo;
    int iKWCode;

    if (!(GetVarPTR (pinfo->pvmGloVars, szName))) { /* New id */

        iKWCode = ((hType == ID_STATE ? KM_STATES /* Translate to KW_ */ /
            : (hType == ID_INPUT ? KM_INPUTS
            : (hType == ID_OUTPUT ? KM_OUTPUTS
            : KM_NULL))));

        if ((hType == ID_PARM) ||
            (hType == (ID_LOCALDYN | ID_SPACEFLAG)) ||
            (hType == (ID_LOCALCALCOUT | ID_SPACEFLAG)) ||
            (hType == (ID_LOCALSCALE | ID_SPACEFLAG))) {
            AddEquation (&pinfo->pvmGloVars, szName, szVal, hType);
            if (hType == ID_PARM)
                printf (" param. '%s' = %s\n", szName, szVal);
        }
        else {
            DeclareModelVar (pibIn, szName, iKWCode);

            /* link value to symbol */
            pvm = GetVarPTR (pinfo->pvmGloVars, szName);
            DefineGlobalVar (pibIn, pvm, szName, szVal, hType);

            if (hType == ID_STATE)
                printf (" species '%s' = %s\n", szName, szVal);

            if (hType == ID_INPUT)
                printf (" input '%s' = %s\n", szName, szVal);

            if (hType == ID_OUTPUT)
                printf (" output '%s' = %s\n", szName, szVal);
        }
    }

} /* SetVar */

/*
-----
Create1Var

    Get the name of the PK template variable stored in pvm. If it starts
    with '_' (underscore) write the name of the SBML species passed in
    pInfo
        at the beginning of the name. Then define the variable.
        This is a callback function for ForAllVar().

*/
int Create1Var (PFILE pfile, PVMMAPSTRCT pvm, PVOID pInfo)

```

```

{
    PFORSV      pV = (PFORSV) pInfo;
    PSTRLEX     szTmp;

    if (pvm->szName[0] == '_') {
        /* extend the variable name with the compartment name */
        sprintf (szTmp, "%s%s", pV->szName, pvm->szName);
        if (pvm->hType == (ID_LOCALDYN | ID_SPACEFLAG))
            SetVar (pV->pibIn, szTmp, pvm->szEqn, pvm->hType);
        else
            SetVar (pV->pibIn, szTmp, pV->szVal, pvm->hType);
    }
    else { /* copy the PK template variable as is */
        if (pvm->hType == ID_INPUT) /* equation undefined, use "0" */
            SetVar (pV->pibIn, pvm->szName, "0", pvm->hType);
        else
            SetVar (pV->pibIn, pvm->szName, pvm->szEqn, pvm->hType);
    }

    return (1);
}

} /* Create1Var */

/*
-----
-----
Transcribe1AlgEqn

Copy with eventual modification an equation from the temporary info
structure of mod (filled in by ReadPKTemplate) to the primary info
structure of mod.

Skip equations of type ID_DERIV.

Transcribe ID_INLINE equations without modifications at all and only
once.

Otherwise:
Get the name of the PK template variable stored in pvm. If it starts
with '_' (underscore) write the name of the SBML species passed in
pInfo
at the beginning of the name. Do the same on all terms of the
equation.

Then register the equation in the primary info structure of mod.

This is a callback function for ForAllVar().

*/
int Transcribe1AlgEqn (PFILE pfile, PVMMAPSTRCT pvm, PVOID pInfo)
{
    PFORSV      pV = (PFORSV) pInfo;
    PSTRLEX     szTmpName = "";
    PSTREQN    szTmpEq = "";
    INPUTBUF    ibDummy;
    PSTRLEX     szLex;
    int         iType;

    if (pvm->hType == ID_INLINE) {
        if (!(GetVarPTR (pV->pTarget, pvm->szName))) { /* New id */

```

```

    DefineVariable (pV->pibIn, pvm->szName, pvm->szEqn, KM_INLINE);
    printf (" inline '%s'\n", pvm->szEqn);
}
return (1);
}

if (pvm->szName[0] == '_') {
    /* extend the variable name with the compartment name */
    sprintf (szTmpName, "%s%s", pV->szName, pvm->szName);
}
else sprintf (szTmpName, "%s", pvm->szName); /* simple copy */

/* deal with the equation */
MakeStringBuffer (NULL, &ibDummy, pvm->szEqn);

while (!EOB(&ibDummy)) {

    NextLex (&ibDummy, szLex, &iType); /* ...all errors reported */

    if ((iType == LX_IDENTIFIER) && !(IsMathFunc (szLex)) &&
        (szLex[0] == '_'))
        sprintf (szTmpEq, "%s%s%s", szTmpEq, pV->szName, szLex); /* extend
*/
    else
        sprintf (szTmpEq, "%s%s", szTmpEq, szLex);

} /* while */

if (!(GetVarPTR (pV->pTarget, szTmpName))) { /* New id */
    if (pvm->hType < ID_DERIV) {
        DefineVariable (pV->pibIn, szTmpName, szTmpEq, KM_NULL);
        printf (" local v. '%s' = %s\n", szTmpName, szTmpEq);
    }
}

return (1);
} /* Transcribe1AlgEqn */

/*
-----
----- Transcribe1DiffEqn

    Copy with eventual modification a differential equation from the
temporary
    info structure of mod (filled in by ReadPKTemplate) to the primary
info
    structure of mod.
    Process only equations of type ID_DERIV.
    Get the name of the PK template variable stored in pvm. If it starts
with '_' (underscore) write the name of the SBML species passed in
pInfo

```

```

at the beginning of the name. Do the same on all terms of the
equation.

Then register the equation in the primary info structure of mod.
This is a callback function for ForAllVar().

*/
int Transcribe1DiffEqn (PFILE pfile, PVMMAPSTRCT pvm, PVOID pInfo)
{
    PFORSV      pV = (PFORSV) pInfo;
    PSTRLEX     szTmpName = "";
    PSTREQN     szTmpEq = "";
    INPUTBUF    ibDummy;
    PSTRLEX     szLex;
    int         iType;

    if ((pvm->hType & ID_TYPEMASK) != ID_DERIV)
        return (0);

    if (pvm->szName[0] == '_') {
        /* extend the variable name with the compartment name */
        sprintf (szTmpName, "%s%s", pV->szName, pvm->szName);
    }
    else sprintf (szTmpName, "%s", pvm->szName); /* simple copy */

    /* deal with the equation */
    MakeStringBuffer (NULL, &ibDummy, pvm->szEqn);

    while (!EOB(&ibDummy)) {

        NextLex (&ibDummy, szLex, &iType); /* ...all errors reported */

        if ((iType == LX_IDENTIFIER) && !(IsMathFunc (szLex)) &&
            (szLex[0] == '_'))
            sprintf (szTmpEq, "%s%s%s", szTmpEq, pV->szName, szLex); /* extend
        */
        else
            sprintf (szTmpEq, "%s%s", szTmpEq, szLex);

    } /* while */

    if (!(GetVarPTR (pV->pTarget, szTmpName))) { /* New id */
        DefineVariable (pV->pibIn, szTmpName, szTmpEq, KM_DXDT);
        printf (" template ODE term for %s = %s\n", szTmpName, szTmpEq);
    }

    return (1);
} /* Transcribe1DiffEqn */

/*
-----
----- ReadCpt
----- Read an SBML compartment tag content. Skip the automatic external

```

```

compartment eventually named by the user. Print the name and size of
the compartment and define it as a global parameter if bTell is TRUE.
*/
void ReadCpt (PINPUTBUF pibIn, ListOf_t *cpt_list, BOOL bTell, long
index)
{
    PSTRLEX szID;
    PSTREQN szEqn;
    PINPUTINFO pinfo = (PINPUTINFO) pibIn->pInfo;
    PVMMAPSTRCT pvm = NULL;
    HANDLE hType;

    Compartmen_t *cpt;

    pinfo->wContext = CN_GLOBAL;

    /* cpt = Compartmen_create (); */
    cpt = (Compartmen_t *) ListOf_get(cpt_list, index);

    sprintf (szID, "%s", Compartmen_getId (cpt));
    sprintf (szEqn, "%lg", Compartmen_getSize (cpt));

    /* exit if compartment has no name */
    if (!strcmp (szID, "(null)")) {
        printf ("***Error: ID field not set for compartment %ld\n", index+1);
        printf ("Exiting...\n\n");
        exit (0);
    }

    /* Ignore the external cpt if the name of that external compartment
       is defined by the user (by setting szDefault_Cpt) when reading the
       SBMLModels section */
    if (!strcmp (szID, szDefault_Cpt)) {
        if (bTell)
            printf (" compart. '%s' = %s (external, ignored)\n", szID, szEqn);
        return;
    }

    if (!(GetVarPTR (pinfo->pvmLocalCpts, szID))) { /* New compartment */

        /* link value to symbol */
        AddEquation (&pinfo->pvmLocalCpts, szID, szEqn, ID_COMPARTMENT);

        if (bTell) {
            /* define it also as a global parameter: Watch out for conflict
with the
               use of a template model... */
            if (!(hType = GetVarType (pinfo->pvmGloVars, szID))) {
                DefineGlobalVar (pibIn, pvm, szID, szEqn, hType);
                printf (" compart. '%s' = %s\n", szID, szEqn);
            }
            else { /* the name was already defined: warning.
                    We could also use pinfo->bTemplateInUse to trap the
template

```

```

        case... */
    printf (" compart. '%s' = %s, redefined, ignored\n", szID,
szEqn);
}
}

} /* end if */

} /* ReadCpt */

/* -----
-----
ReadCpts

    Read an SBML list of compartments section.

*/
void ReadCpts (PINPUTBUF pibIn, Model_t *model, BOOL bTell)
{
    PINPUTINFO pinfo = (PINPUTINFO) pibIn->pInfo;
    long i, n;
    ListOf_t *cpt_list;

    /* reset the compartment list for that SBML file */
    pinfo->pvmLocalCpts = NULL;

    cpt_list = Model_getListOfCompartments (model);
    n = ListOf_size (cpt_list);

    if (bTell)
        printf( "\n number of compartments: %ld\n", n);

    for (i = 0; i < n; i++)
        ReadCpt (pibIn, cpt_list, bTell, i);

} /* ReadCpts */

/* -----
-----
ReadFunction

    Read a function definition section in the given SBML buffer. Functions
are
    translated as C preprocessor's macros.

*/
void ReadFunction (PINPUTBUF pibIn, ListOf_t *functions_list, long index)
{
    PSTRLEX szFName;
    PSTREQN szEqn, szMacro = "";
    PINPUTINFO pinfo = (PINPUTINFO) pibIn->pInfo;
    FunctionDefinition_t *FunctionX;
    int i, nArg;

```

```

/* set context to Dynamics section */
pinfo->wContext = CN_DYNAMICS;

FunctionX = (FunctionDefinition_t *) ListOf_get (functions_list,
index);
sprintf (szFName, "%s", FunctionDefinition_getId (FunctionX));

if (!strcmp (szFName, "(null)")) {
    printf ("***Error: ID field not set for function %ld\n", index+1);
    printf ("Exiting...\n\n");
    exit (0);
}

/* start writing the macro */
sprintf (szMacro, "#define %s(", szFName);

/* get the number of parameters */
nArg = FunctionDefinition_getNumArguments (FunctionX);

/* get parameters, continue writing the macro */
for (i = 0; i < nArg; i++)
    sprintf (szMacro, "%s%s%", szMacro, SBML_formulaToString (
        FunctionDefinition_getArgument (FunctionX, i)),
        (i < nArg - 1 ? "," : ")"));

/* read formula */
sprintf (szEqn, "%s",
    SBML_formulaToString (FunctionDefinition_getBody
(FunctionX)));

printf (" function %s = %s\n", szMacro, szEqn);

/* finish writing the macro */
sprintf (szMacro, "%s (%s)", szMacro, szEqn);

/* define reaction name as in line spec in the Dynamics section */
DefineVariable (pibIn, "Inline", szMacro, KM_INLINE);
DefineVariable (pibIn, szFName, "", KM_FUNCTION);

} /* ReadFunction */

/*
-----
----- ReadFunctions

    Read a list of function definitions section.
*/
void ReadFunctions (PINPUTBUF pibIn, int iSBML_level, Model_t *model)
{
    long i, nF;
    ListOf_t *functions_list;

    functions_list = Model_getListOfFunctionDefinitions(model);

```

```

nF = ListOf_size (functions_list);

if (nF != 0) {
    printf ("\n number of functions: %ld\n" , nF);

    for (i = 0; i < nF; i++)
        ReadFunction (pibIn, functions_list, i);
}

} /* ReadFunctions */

/*
-----
-----  

ReadDifferentials

Construct a differential equation for each species involved in the
reactions of the SBML model stored in the given buffer.

*/
void ReadDifferentials (PINPUTBUF pibIn, Model_t *model)
{
    PSTRLEX szRName;
    PINPUTINFO pinfo = (PINPUTINFO) pibIn->pInfo;

    /* set context to Dynamics section */
    pinfo->wContext = CN_DYNAMICS;

    ListOf_t *reaction_list, *reactant_list, *product_list, *modifier_list;
    Reaction_t *reaction;
    long r, r2, i, j;
    SpeciesReference_t *speciesref;

    /* get reaction list */
    reaction_list = Model_getListOfReactions(model);
    r = ListOf_size (reaction_list);

    printf ("\n number of reactions: %ld\n" , r);

    for (i = 0; i < r; i++) { /* for each reaction in the model */

        /* get reaction name */
        reaction = (Reaction_t *) ListOf_get (reaction_list, i);
        sprintf (szRName, "%s", Reaction_getId (reaction));

        printf ("\n reaction ID: %s\n", szRName);

        /* get the reactants' list */
        reactant_list = Reaction_getListOfReactants (reaction);
        r2 = ListOf_size (reactant_list);

        printf (" number of reactants: %ld\n" , r2);
    }
}

```

```

for (j = 0; j < r2; j++) { /* for each reactant */

    /* get species reference ID */
    speciesref =(SpeciesReference_t *) ListOf_get (reactant_list, j);

    /* construct the differential for that reactant */
    ConstructEqn (pibIn, szRName, speciesref, reactant);
}

/* get the products' list */
product_list = Reaction_getListOfProducts (reaction);
r2 = ListOf_size (product_list);

printf (" number of products: %ld\n" , r2);

/* for each product */
for (j = 0; j < r2; j++) { /* for each product */

    /* get species reference ID */
    speciesref =(SpeciesReference_t *) ListOf_get (product_list, j);

    /* construct the differential for that product */
    ConstructEqn (pibIn, szRName, speciesref, product);
}

/* get the modifiers' list */
modifier_list = Reaction_getListOfModifiers (reaction);
r2 = ListOf_size (modifier_list);

printf (" number of modifiers: %ld\n" , r2);

/* for each modifier */
for (j = 0; j < r2; j++) { /* for each product */

    /* get species reference ID */
    speciesref =(SpeciesReference_t *) ListOf_get (modifier_list, j);

    /* construct the differential for that product */
    ConstructEqn (pibIn, szRName, speciesref, modifier);
}

} /* end while */

} /* ReadDifferentials */

/*
-----
----- ReadParameter

Read an SBML global parameter tag content and set it up as global
variable.
*/
void ReadParameter (PINPUTBUF pibIn, ListOf_t *param_list, long index)

```

```

{
    PSTRLEX szID;
    PSTREQN szEqn;
    PVMMAPSTRCT pvm = NULL;
    HANDLE hType;
    PINPUTINFO pinfo = (PINPUTINFO) pibIn->pInfo;
    Parameter_t *param;
    pinfo->wContext = CN_GLOBAL;

    param =(Parameter_t *) ListOf_get (param_list, index);
    sprintf (szID, "%s", Parameter_getId (param));

    if (!strcmp (szID, "(null)")) {
        printf ("***Error: ID field not set for parameter %ld\n", index+1);
        printf ("Exiting...\n\n");
        exit (0);
    }

    if (!(hType = GetVarType (pinfo->pvmGloVars, szID))) { /* New ID */
        sprintf (szEqn, "%g", Parameter_getValue (param));
        /* link value to symbol */
        DefineGlobalVar (pibIn, pvm, szID, szEqn, hType);

        printf (" param. '%s' = %s\n", szID, szEqn);
    } /* end if */

    else { /* the parameter was already defined, this is confusing, exit */
        printf ("***Error: redeclaration of parameter %s\n", szID);
        printf ("Exiting...\n\n");
        exit (0);
    }
}

} /* ReadParameter */

/*
-----
----- ReadParameters

    Read an SBML list of parameters section.
*/
void ReadParameters (PINPUTBUF pibIn, Model_t *model)
{
    long p, i;
    ListOf_t *param_list;

    param_list = Model_getListOfParameters(model);
    p = ListOf_size (param_list);

    printf ("\n number of parameters: %ld\n" , p);
}

```

```

    for (i = 0; i < p; i++) {
        ReadParameter(pibIn, param_list, i);
    }

} /* ReadParameters */

/*
-----
----- ReadReaction_L1
    Read an SBML reaction tag in the given SBML level 1 buffer.
*/
void ReadReaction_L1 (PINPUTBUF pibIn, ListOf_t *reaction_list, long index)
{
    PSTRLEX szRName;
    PSTREQN szEqn;
    PINPUTINFO pinfo = (PINPUTINFO) pibIn->pInfo;

    Reaction_t *reaction;
    KineticLaw_t *kl;

    /* set context to Dynamics section */
    pinfo->wContext = CN_DYNAMICS;

    reaction = (Reaction_t *) ListOf_get (reaction_list, index);
    sprintf (szRName, "%s", Reaction_getId (reaction));

    if (!strcmp (szRName, "(null)")) {
        printf ("***Error: ID field not set for reaction %ld\n", index+1);
        printf ("Exiting...\n\n");
        exit (0);
    }

    /* get kinetic law section */
    kl = Reaction_getKineticLaw (reaction);
    sprintf (szEqn, "%s", KineticLaw_getFormula (kl));

    /* define reaction name as a local variable in the Dynamics section */
    DefineVariable (pibIn, szRName, szEqn, 0);

} /* ReadReaction_L1 */

/*
-----
----- ReadReaction_L2
    Read a reaction tag in the given SBML level 2 buffer.
*/
void ReadReaction_L2 (PINPUTBUF pibIn, ListOf_t *reaction_list, long index)
{

```

```

PSTRLEX      szRName, szLex;
PSTREQN      szEqn, szEqn_expanded = "";
PINPUTINFO   pinfo = (PINPUTINFO) pibIn->pInfo;
INPUTBUF     bufSzEqn;
PINPUTBUF    pbufSzEqn = &bufSzEqn;
long         i, p;
int          iType;

Reaction_t   *reaction;
KineticLaw_t *kl;
ListOf_t     *param_list;

reaction = (Reaction_t *) ListOf_get (reaction_list, index);

sprintf (szRName, "%s", Reaction_getId (reaction));

/* check if reaction name is valid */
if (!strcmp (szRName, "(null)") ) {
    printf ("***Error: ID field not set for reaction %ld\n", index+1);
    printf ("Exiting...\n\n");
    exit (0);
}

/* print reaction name to the screen */
printf ("\n reaction ID: %s\n", szRName);

kl = Reaction_getKineticLaw (reaction);

/* get local parameter list first */
param_list = KineticLaw_getListOfParameters(kl);
p = ListOf_size (param_list);

printf (" number of local parameters (made global by MCSim): %ld\n",
p);

for (i = 0; i < p; i++)
    ReadParameter(pibIn, param_list, i); /* side effect!
                                         sets the context to GLOBAL */

/* get kinetic law section */
sprintf (szEqn, "%s", KineticLaw_getFormula (kl));

printf (" '%s' = %s\n", szRName, szEqn);

/* if a PK template is used, reactions are supposed to happen in the
   compartment in which they are defined: scan szEqn and
   pad species name with that compartment name */
if (pinfo->bTemplateInUse) {
    /* link szEqn in a temporary buffer for parsing */
    MakeStringBuffer (NULL, pbufSzEqn, szEqn);

    while (!EOB(pbufSzEqn)) {

        NextLex (pbufSzEqn, szLex, &iType); /* ...all errors reported */

```

```

    if ((iType == LX_IDENTIFIER ) && !(IsMathFunc (szLex))) {
        if (!GetVarPTR (pinfo->pvmGloVars, szLex))
            sprintf (szLex, "%s_%s", szLex, pinfo->pvmLocalCpts->szName);
    }

    sprintf (szEqn_expanded, "%s %s", szEqn_expanded, szLex);
}

/* while */

printf (" after template processing:\n");
printf (" '%s' = %s\n", szRName, szEqn_expanded);

} /* if PK template */

/* set context to Dynamics section */
pinfo->wContext = CN_DYNAMICS;

/* define reaction name as a local variable in the Dynamics section */
DefineVariable (pibIn, szRName,
                ((pinfo->bTemplateInUse) ? szEqn_expanded : szEqn), 0);

} /* ReadReaction_L2 */

/*
-----
----- ReadReactions

    Read a list of reactions sections.
*/
void ReadReactions (PINPUTBUF pibIn, int iSBML_level, Model_t *model)
{
    long r, i;
    ListOf_t *reaction_list;

    reaction_list = Model_getListOfReactions(model);
    r = ListOf_size (reaction_list);

    printf ("\n number of reactions: %ld\n" , r);

    for (i = 0; i < r; i++) {

        if (iSBML_level == 1)
            ReadReaction_L1 (pibIn, reaction_list, i);
        else
            ReadReaction_L2 (pibIn, reaction_list, i);
    }
}

} /* ReadReactions */

```

```

/*
-----
----- ReadRule

    Read a rate or assignment rule section in the given SBML buffer.
*/
void ReadRule (PINPUTBUF pibIn, ListOf_t *rules_list, long index)
{
    PSTRLEX szRName;
    PSTREQN szEqn;
    PINPUTINFO pinfo = (PINPUTINFO) pibIn->pInfo;

    Rule_t *rule;

    rule = (Rule_t *) ListOf_get (rules_list, index);

    if (Rule_isAlgebraic (rule))
        printf ("*** Warning: algebraic rule is ignored. ***\n");
    else {
        if (Rule_isAssignment (rule)) {

            /* assignment rules are valid for boundary species, defined by MCSim
            as
                parameters, for compartment volumes and parameters.
                Set context to the Scale section */
            pinfo->wContext = CN_SCALE;

            sprintf (szRName, "%s", Rule_getVariable (rule));
            sprintf (szEqn, "%s", Rule_getFormula (rule));

            /* define assignment rule as an assignment in Scale section */
            DefineVariable (pibIn, szRName, szEqn, 0);
        }
        else {
            if (Rule_isRate (rule)) {

                /* set context to Dynamics section */
                pinfo->wContext = CN_DYNAMICS;

                sprintf (szRName, "%s", Rule_getVariable (rule));
                sprintf (szEqn, "%s", Rule_getFormula (rule));

                /* define rate rule as Derivative spec in the Dynamics section */
                DefineVariable (pibIn, szRName, szEqn, KM_DXDT);
            }
        }
        printf (" rule %s = %s\n", szRName, szEqn);
    }
} /* ReadRule */

-----
-----
```

```

ReadRules

    Read a list of rate or assignment rules.
*/
void ReadRules (PINPUTBUF pibIn, int iSBML_level, Model_t *model)
{

    long ru, i;
    ListOf_t *rules_list;

    rules_list = Model_getListOfRules(model);
    ru = ListOf_size (rules_list);

    printf ("\n number of rules: %ld\n" , ru);

    for (i = 0; i < ru; i++){
        if (iSBML_level == 1)
            printf ("mod: ignoring rate rules definitions in level 1...\n");
        else
            ReadRule (pibIn, rules_list, i);
    }

} /* ReadRules */

/* -----
----- ReadSBMLLevel

    Read a sbml tag content and get the level.
*/
int ReadSBMLLevel (SBMLDocument_t *d)
{
    int level;

    level = SBMLDocument_getLevel(d);

    printf ("sbml level %d\n", level);

    return (level);

} /* ReadSBMLLevel */

/* -----
----- Read1Species

    Read a species tag content. Set it up as state variables.
    If a PK template is used, process also all the automatic PK variables
to
    be created from that species.
    If bProcessPK_ODEs is TRUE, the derivatives specified by the PK
template

```

```

    are processed and copied to the main info.
*/
void Read1Species (PINPUTBUF pibIn, BOOL bProcessPK_ODEs, ListOf_t
*species_list, long index)
{
    PSTRLEX      szID;
    PSTRLEX      szCpt;
    PSTREQN     szEqn;
    BOOL        bBoundary;
    FORSV       sVar;
    HANDLE       hType;
    PINPUTINFO   pinfo = (PINPUTINFO) pibIn->pInfo;
    PINPUTINFO   ptempinfo = (PINPUTINFO) pibIn->pTempInfo;
    PVMMAPSTRCT pvm = NULL;

    pinfo->wContext = CN_GLOBAL;
    Species_t *species;

    species = (Species_t *) ListOf_get (species_list, index);

    switch (iIDtype) {
        case ID:
            sprintf (szID, "%s", Species_getId (species));
            break;
        case name:
            sprintf (szID, "%s", Species_getName (species));
            break;
        case metaID:
            sprintf (szID, "%s", SBase_getMetaId ((SBase_t *) species));
            break;
    }

    if (Species_getHasOnlySubstanceUnits (species)) {
        if (Species_isSetInitialAmount (species))
            sprintf (szEqn, "%g", Species_getInitialAmount (species));
        else {
            printf ("***Error: Species has only substance units but "
                   "InitialAmount is not set.\n"
                   "Exiting.\n\n");
            exit (0);
        }
    }
    else {
        if (Species_isSetInitialConcentration (species))
            sprintf (szEqn, "%g", Species_getInitialConcentration (species));
        else {
            sprintf (szEqn, "%g", Species_getInitialAmount (species));
            printf ("\nWarning: Species should be concentration but "
                   "InitialConcentration is not set.\n\n");
        }
    }

    sprintf (szCpt, "%s", Species_getCompartment (species));
    bBoundary = (BOOL) Species_getBoundaryCondition (species);
}

```

```

if (!strcmp (szID, "(null)")) {
    printf ("***Error: ID field not set for species %ld\n", index + 1);
    printf ("Exiting...\n\n");
    exit (0);
}

if (pinfo->bTemplateInUse) {

    printf ("\n template processing for species '%s':\n", szID);

    /* reset the species' value, to avoid confusion in case of
    redefinition */
    sprintf (szEqn, "0");

    /* check if the compartment is the external default or not */
    if (strcmp (szCpt, szDefault_Cpt)) {

        printf (" (species local to compartment '%s')\n", szCpt);

        /* species is in a template-defined cpt */
        if (!(GetVarPTR (ptempinfo->pvmCpts, szCpt))) {
            /* compartment not defined by the template: error */
            printf ("***Error: template did not define");
            printf (" compartment '%s' - exiting...\n\n", szCpt);
            exit (0);
        }
        else /* extend the variable name with the compartment name */
            sprintf (szID, "%s_%s", szID, szCpt);

        if (bBoundary) {
            /* species assigned boundary conditions are defined as parameters
*/
            if (!(hType = GetVarType (pinfo->pvmGloVars, szID))) { /* New id
*/
                /* link value to symbol */
                DefineGlobalVar (pibIn, pvm, szID, szEqn, hType);
                printf (" param. '%s' = %s (was boundary species)\n", szID,
szEqn);
                } /* end if */
            } /* end if bBoundary */
        else /* not boundary, create a state variable */
            SetVar (pibIn, szID, szEqn, ID_STATE);
    } /* if */

    else { /* species is outside of a template-defined compartment */

        printf (" (circulating species)\n");

        /* first: species set to boundary conditions (i.e. invariant) are
        not allowed to circulate. If found outside of a meaningful
        compartment: exit with error message */
        if (bBoundary) {
            printf ("***Error: Species %s is set to boundary;\n", szID);
        }
    }
}

```

```

        printf ("           It has to be inside a meaningful compartment -\n");
    );
        printf ("exiting.\n\n");
        exit (0);
    }

/* species in external compartment, store its info to pass through
   ForAllVar list scanning routine */
sVar.pibIn  = pibIn;
sVar.szName = szID;
sVar.szVal  = szEqn;

/* create state variables, input, outputs and parameters, adding
the
   species name at the beginning of each state variable of the
   template that starts with '_' */
ForAllVar (NULL, ptempinfo->pvmGloVars,
           &Create1Var, ID_NULL, (PVOID) &sVar);

/* same for the PK variables local to Dynamics */
pinfo->wContext = CN_DYNAMICS;
sVar.pTarget = pinfo->pvmDynEqns;
ForAllVar (NULL, ptempinfo->pvmGloVars,
           &Create1Var, ID_LOCALDYN, (PVOID) &sVar);
/* transcribe PK dynamic equations, except derivatives */
ForAllVar (NULL, ptempinfo->pvmDynEqns,
           &Transcribe1AlgEqn, ID_NULL, (PVOID) &sVar);
/* transcribe derivatives only if requested */
if (bProcessPK_ODEs)
    ForAllVar (NULL, ptempinfo->pvmDynEqns,
               &Transcribe1DiffEqn, ID_NULL, (PVOID) &sVar);

/* same for Scale */
pinfo->wContext = CN_SCALE;
sVar.pTarget = pinfo->pvmScaleEqns;
ForAllVar (NULL, ptempinfo->pvmGloVars,
           &Create1Var, ID_LOCALSCALE, (PVOID) &sVar);
ForAllVar (NULL, ptempinfo->pvmScaleEqns,
           &Transcribe1AlgEqn, ID_NULL, (PVOID) &sVar);

/* same CalcOutputs */
pinfo->wContext = CN_CALCOUTPUTS;
sVar.pTarget = pinfo->pvmCalcOutEqns;
ForAllVar (NULL, ptempinfo->pvmGloVars,
           &Create1Var, ID_LOCALCALCOUT, (PVOID) &sVar);
ForAllVar (NULL, ptempinfo->pvmCalcOutEqns,
           &Transcribe1AlgEqn, ID_NULL, (PVOID) &sVar);
}

} /* end if (pinfo->bTemplateInUse) */

else { /* no PK template, process the variable, ignoring compartments
*/
    if (bBoundary) {

```

```

    /* species assigned boundary conditions are defined as parameters
 */
    if (!(hType = GetVarType (pinfo->pvmGloVars, szID))) { /* New id */
        /* link value to symbol */
        DefineGlobalVar (pibIn, pvm, szID, szEqn, hType);
        printf (" param. '%s' = %s (was boundary species)\n", szID,
szEqn);
    } /* end if */
} /* end if bBoundary */
else /* not boundary, create a state variable */
    SetVar (pibIn, szID, szEqn, ID_STATE);
}

} /* Read1Species */

/*
-----
----- ReadSpecies

    Read a list of species section.
*/
void ReadSpecies (PINPUTBUF pibIn, int iSBML_level, BOOL bProcessPK_ODEs,
                  Model_t *model)
{
    long p, i;
    ListOf_t *species_list;

    species_list = Model_getListOfSpecies (model);
    p = ListOf_size (species_list);

    printf ("\n number of original SBML species: %ld\n" , p);

    for (i = 0; i < p; i++) {
        Read1Species (pibIn, bProcessPK_ODEs, species_list, i);
    }

} /* ReadSpecies */

/*
-----
----- ReadFileNames

    Reads a list of strings giving the names of the SBML files to process.
*/
void ReadFileNames (PINPUTBUF pibIn, PLONG nFiles, PSTR **pszNames)
{
    long i;
    int iLexType, iErr = 0;
    char szLex[MAX_FILENAME_SIZE];
    PSTRLEX szPunct;
    PSTR pbufStore;

```

```

/* store current buffer position */
pbufStore = pibIn->pbufCur;

do { /* get number of model filenames in list */
    GetaString (pibIn, szLex);
    *nFiles = *nFiles + 1;
    NextLex (pibIn, szPunct, &iLexType);
    SkipWhitespace (pibIn);

    if (!(iLexType & LX_IDENTIFIER)) {
        /* not an identifier, should be ',' or ')' */
        if ((szPunct[0] != ',') && (szPunct[0] != CH_RBRACE))
            iErr = szPunct[1] = CH_RBRACE;
    }
} while ((szPunct[0] != CH_RBRACE) && (!iErr));

if (!(*pszNames = (PSTR *) malloc (*nFiles * sizeof(PSTR))))
    ReportError (NULL, RE_OUTOFMEM | RE_FATAL, "ReadJModels", NULL);

/* get the actual filenames */
pibIn->pbufCur = pbufStore;
for (i = 0; i < *nFiles; i++) {
    GetaString (pibIn, szLex);
    NextLex (pibIn, szPunct, &iLexType);
    SkipWhitespace (pibIn);

    if ( !((*pszNames)[i] = (PSTR) malloc (strlen (szLex) + 1)))
        ReportError (NULL, RE_OUTOFMEM | RE_FATAL, "ReadFileNames", NULL);
    else
        strcpy ((*pszNames)[i], szLex);
} /* for i */

} /* ReadFileNames */

/*
-----
----- ReadOptions

    Reads tokens indicating user-specified general model processing
options.
    These are optional, but if one is given they must all be given. The
list of
        SBML files starts after.
*/
void ReadOptions (PINPUTBUF pibIn)
{
    int iLexType;
    PSTRLEX szLex, szPunct;

    /* if a double quote is first met assume that no options are given and
       use defaults */
    SkipWhitespace (pibIn);
    if (*pibIn->pbufCur == '') {

```

```

    iIDtype = ID; /* default because IDs are used in reactions */
    strcpy (szDefault_Cpt, "default");
}
else {

/* get UseName, UseID or UseMetaID option */
NextLex (pibIn, szLex, &iLexType);

/* switch according to szLex */
if (!strcmp (szLex, "UseID"))
    {iIDtype = ID;      goto Done;}
if (!strcmp (szLex, "UseName")) /* is problematic for reactions */
    {iIDtype = name;   /* goto Done; */}
if (!strcmp (szLex, "UseMetaID")) /* is problematic for reactions */
    {iIDtype = metaID; /* goto Done; */}
/* if we arrive here there is a problem */
printf ("***Error: SBMLModel 1st option must be 'UseID'."
        "'UseName' or 'UseMetaID' are disabled for now.\n"
        "Exiting.\n\n");
exit(0);

Done:

NextLex (pibIn, szPunct, &iLexType);
SkipWhitespace (pibIn);

/* get external compartment name, should be between double quotes */
GetAString (pibIn, szLex);

/* assign to global variable */
strcpy (szDefault_Cpt, szLex);

NextLex (pibIn, szPunct, &iLexType);
SkipWhitespace (pibIn);
}

} /* ReadOptions */

/*
-----
----- ReadSBMLModels

    Read the list of SBML model definition files given in an SBMLModels
list.
*/
void ReadSBMLModels (PINPUTBUF pibIn)
{
    long i, nFiles = 0;
    PSTR *pszFileNames = NULL;
    INPUTBUF ibInLocal;
    int iSBML_level;
    PINPUTINFO pinfo = (PINPUTINFO) pibIn->pInfo;
    SBMLDocument_t *document;
}

```

```

Model_t           *model;

/* read the options at start of the model list */
ReadOptions (pibIn);

/* read the SBML model file names from current buffer */
ReadFileNames (pibIn, &nFiles, &pszFileNames);

/* in each file, get the functions, the compartments, variables and
   rate rules or reactions (to be set up as local variables) */
for (i = 0; i < nFiles; i++) {

    printf ("\nreading model '%s'\n", pszFileNames[i]);

    /* init buffer and read in the input file. */
    /* buffer size -1 will create a buffer of the size of the input file
*/
    if (!InitBuffer (&ibInLocal, -1, pszFileNames[i]))
        ReportError (&ibInLocal, RE_INIT | RE_FATAL, "ReadSBMLModels",
NULL);

    /* attach info records to input buffer */
    ibInLocal.pInfo = pibIn->pInfo;
    ibInLocal.pTempInfo = pibIn->pTempInfo;

    document = readSBML (pszFileNames[i]);

    SBMLDocument_printErrors (document, stdout);

    model = SBMLDocument_getModel (document);
    if (model == NULL) {
        printf ("***Error: No model present. Exiting.\n\n");
        exit (0);
    }

    /* read the SBML level */
    iSBML_level = ReadSBMLLevel (document);

    /* PK template requires level 2 SBML, issue an error otherwise */
    if ((pinfo->bTemplateInUse) && (iSBML_level < 2)) {
        printf ("***Error: use of a PK template requires ");
        printf ("SBML level 2 - exiting.\n\n");
        exit (0);
    }

    /* read compartments, do not presume order, reset buffer */
    ibInLocal.pbufCur = ibInLocal.pbufOrg;
    ReadCpts (&ibInLocal, model, TRUE); /* TRUE -> print the cpt name
etc. */

    /* read function definitions, reset buffer */
    ibInLocal.pbufCur = ibInLocal.pbufOrg;
    ReadFunctions (&ibInLocal, iSBML_level, model);
}

```

```

/* read global parameters, reset buffer */
ibInLocal.pbufCur = ibInLocal.pbufOrg;
ReadParameters (&ibInLocal, model);

/* read SBML species, reset buffer */
ibInLocal.pbufCur = ibInLocal.pbufOrg;
ReadSpecies (&ibInLocal, iSBML_level, FALSE, /* don't bProcessPK_ODEs
*/
model);

/* read SBML rate rules, reset buffer */
ibInLocal.pbufCur = ibInLocal.pbufOrg;
ReadRules (&ibInLocal, iSBML_level, model);

/* read SBML reactions, reset buffer */
ibInLocal.pbufCur = ibInLocal.pbufOrg;
ReadReactions (&ibInLocal, iSBML_level, model);

} /* for model index i*/

/* now scan again to read the differential equations */
for (i = 0; i < nFiles; i++) {

/* init buffer and read in the input file. */
if (!InitBuffer (&ibInLocal, -1, pszFileNames[i]))
    ReportError (&ibInLocal, RE_INIT | RE_FATAL, "ReadJModels", NULL);

ibInLocal.pInfo = pibIn->pInfo;
ibInLocal.pTempInfo = pibIn->pTempInfo;

/* reinitialize model */
document = readSBML(pszFileNames[i]);
model = SBMLDocument_getModel (document);

/* re-read compartments, no printing, no redeclaration */
ReadCpts (&ibInLocal, model, FALSE);

printf ("\nreading differentials in model %s\n", pszFileNames[i]);

/* re-read SBML species, reset buffer */
ibInLocal.pbufCur = ibInLocal.pbufOrg;
ReadSpecies (&ibInLocal, iSBML_level, TRUE, /* TRUE: bProcessPK_ODEs
*/
model);

ReadDifferentials (&ibInLocal, model);

} /* for model index i*/

printf ("\n");

/* cleanup */
for (i = 0; i < nFiles; i++)
    free (pszFileNames[i]);

```

```

    free (pszFileNames);

    pInfo->wContext = CN_END;

} /* ReadSBMLModels */

/*
-----
----- ReadPKTemplate

Read the template pharmacokinetic model definition in the file given
by the next lexical element of the buffer argument. Information is
stored in the pTmpInfo structure of the pibIn buffer argument.

*/
void ReadPKTemplate (PINPUTBUF pibIn)
{
    INPUTBUF ibInLocal;
    PSTRLEX szLex; /* Lex elem of MAX_LEX length */
    PSTREQN szEqn; /* Equation buffer of MAX_EQN length */
    int iLexType;
    long nFiles = 0;
    PSTR *pszFileNames;
    PINPUTINFO pInfo;

    /* exchange info and tempInfo pointers to store data in fact in
     tempInfo */
    pInfo = (PINPUTINFO) pibIn->pTempInfo;

    /* reset context */
    pInfo->wContext = CN_GLOBAL;

    /* read the template model file name from current buffer */
    ReadFileNames (pibIn, &nFiles, &pszFileNames);

    if (nFiles > 1)
        printf ("mod: cannot use more than one template - using only the
1st\n\n");

    /* give the template name used */
    printf ("%s\n", pszFileNames[0]);

    if (!InitBuffer (&ibInLocal, BUFFER_SIZE, pszFileNames[0]))
        ReportError (&ibInLocal, RE_INIT | RE_FATAL, "ReadModel", NULL);

    ibInLocal.pInfo = (PVOID) pInfo; /* Attach info to local input buffer
*/
    do { /* State machine for parsing syntax */
        NextLex (&ibInLocal, szLex, &iLexType);
        switch (iLexType) {
            case LX_NULL:

```

```

    pinfo->wContext = CN_END;
    break;

    case LX_IDENTIFIER:
        ProcessWord (&ibInLocal, szLex, szEqn);
        break;

    case LX_PUNCT: case LX_EQNPUNCT:
        if (szLex[0] == CH_STMTTERM) {
            break;
        }
        else {
            if (szLex[0] == CH_RBRACE &&
                (pinfo->wContext & (CN_DYNAMICS | CN_JACOB | CN_SCALE))) {
                pinfo->wContext = CN_GLOBAL;
                break;
            }
            else {
                if (szLex[0] == CH_COMMENT) {
                    SkipComment (&ibInLocal);
                    break;
                }
                /* else: fall through! */
            }
        }
    }

default:
    ReportError (&ibInLocal, RE_UNEXPECTED, szLex, "* Ignoring");
    break;

case LX_INTEGER:
case LX_FLOAT:
    ReportError (&ibInLocal, RE_UNEXPNUMBER, szLex, "* Ignoring");
    break;

} /* switch */

} while (pinfo->wContext != CN_END &&
        (*ibInLocal.pbufCur ||
         FillBuffer (&ibInLocal, BUFFER_SIZE) != EOF));

fclose (ibInLocal.pfileIn);

ReversePointers (&pinfo->pvmGloVars);
ReversePointers (&pinfo->pvmDynEqns);
ReversePointers (&pinfo->pvmScaleEqns);
ReversePointers (&pinfo->pvmCalcOutEqns);
ReversePointers (&pinfo->pvmJacobEqns);

/* reset pinfo pointers */
pinfo = (PINPUTINFO) pibIn->pInfo;

/* set context */
pinfo->wContext = CN_TEMPLATE_DEFINED;

```

```
pinfo->bTemplateInUse = TRUE;  
} /* ReadPKTemplate */  
#endif /* HAVE_LIBSBML */
```